Combinatorial testing

Learning objectives

- Understand rationale and basic approach for systematic combinatorial testing
- Learn how to apply some representative combinatorial approaches
  - Category-partition testing
  - Pairwise combination testing
  - Catalog-based testing
- Understand key differences and similarities among the approaches
  - and application domains for which they are suited

Combinatorial testing: Basic idea

- Identify **distinct attributes** that can be varied
  - In the data, environment, or configuration
  - Example: browser could be “Chrome” or “Firefox”, operating system could be “Windows OS”, “MacOS”, or “Android OS”
- Systematically generate **combinations** to be tested
  - Example: Chrome on Windows OS, Chrome on Mac OS, Firefox on Windows OS, Firefox on Mac OS, …
- Rationale: Test cases should be varied and include possible “corner cases”

Key ideas in combinatorial approaches

- Category-partition testing
  - separate (manual) identification of values that characterize the input space from (automatic) generation of combinations for test cases
- Pairwise testing
  - systematically test interactions among attributes of the program input space with a relatively small number of test cases
- Catalog-based testing
  - aggregate and synthesize the experience of test designers in a particular organization or application domain, to aid in identifying attribute values

Category partition (manual steps)

1. Decompose the specification into independently **testable features**
   - for each feature identify
     - **parameters**
     - **environment elements**
   - for each parameter and environment element identify elementary characteristics (**categories**)
2. Identify relevant values
   - for each characteristic (category) identify (classes of) values
     - normal values
     - boundary values
     - special values
     - error values
3. Introduce constraints

An informal specification: *check configuration*

Check Configuration

- Check the validity of a computer configuration
- The parameters of check-configuration are:
  - Model
  - Set of components

An informal specification: parameter *model*

Model

- A model identifies a specific product and determines a set of constraints on available components. Models are characterized by logical slots for components, which may or may not be implemented by physical slots on a bus. Slots may be required or optional. Required slots must be assigned with a suitable component to obtain a legal configuration, while optional slots may be left empty or filled depending on the customers' needs

Example:

The required "slots" of the Chipmunk C20 laptop computer include a screen, a processor, a hard disk, memory, and an operating system. (Of these, only the hard disk and memory are implemented using actual hardware slots on a bus.) The optional slots include external storage devices such as USB.

An informal specification of parameter *set of components*

Set of Components

- A set of *(slot, component)* pairs, corresponding to the required and optional slots of the model. A *component* is a choice that can be varied within a model, and which is not designed to be replaced by the end user. Available components and a default for each slot is determined by the model. The special value *empty* is allowed (and may be the default selection) for optional slots. In addition to being compatible or incompatible with a particular model and slot, individual components may be compatible or incompatible with each other.

Example:

The default configuration of the Chipmunk C20 includes 20 gigabytes of hard disk; 30 and 40 gigabyte disks are also available. (Since the hard disk is a required slot, empty is not an allowed choice.) The default operating system is RodentOS 3.2, personal edition, but RodentOS 3.2 mobile server edition may also be selected. The mobile server edition requires at least 30 gigabytes of hard disk.

Step1: Identify independently testable units and categories

- Choosing categories
  - no hard-and-fast rules for choosing categories
  - not a trivial task!
- Categories reflect test designer's judgment
  - regarding which classes of values may be treated differently by an implementation
- Choosing categories well requires experience and knowledge
  - of the application domain and product architecture. The test designer must look under the surface of the specification and identify hidden characteristics

Step 1: Identify independently testable units and categories

Parameter *Model*

- Model number
- Number of required slots for selected model (#SMRS)
- Number of optional slots for selected model (#SMOS)

Parameter *Components*

- Correspondence of selection with model slots
- Number of required components with non-empty selection
- Required component selection
- Number of optional components with non-empty selection
- Optional component selection

Environment element: *Product database*

- Number of models in database (#DBM)
- Number of components in database (#DBC)

Step 2: Identify relevant values

- Identify (list) representative classes of values for each of the categories
  - Ignore ***interactions*** among values for different categories (considered in the next step)
- Representative values may be identified by applying
  - Boundary value testing
    - select extreme values within a class
    - select values outside but as close as possible to the class
    - select interior (non-extreme) values of the class
  - Erroneous condition testing
    - select values outside the normal domain of the program

Step 2: Identify relevant values: Model

- Model number
  - Malformed
  - Not in database
  - Valid
- Number of required slots for selected model (#SMRS)
  - 0
  - 1
  - Many
- Number of optional slots for selected model (#SMOS)
  - 0
  - 1
  - Many

Step 2: Identify relevant values: Components

- Correspondence of selection with model slots
  - Omitted slots
  - Extra slots
  - Mismatched slots
  - Complete correspondence
- Number of required components with non empty selection
  - 0
  - < number required slots
  - = number required slots
- Required component selection
  - Some defaults
  - All valid
  - ≥ 1 incompatible with slots
  - ≥ 1 incompatible with another selection
  - ≥ 1 incompatible with model
  - ≥ 1 not in database
- Number of optional components with non empty selection
  - 0
  - < #SMOS
  - = #SMOS
- Optional component selection
  - Some defaults
  - All valid
  - ≥ 1 incompatible with slots
  - ≥ 1 incompatible with another selection
  - ≥ 1 incompatible with model
  - ≥ 1 not in database

Step 2: Identify relevant values: Database

Number of models in database (#DBM)

- 0
- 1
- Many

Number of components in database (#DBC)

- 0
- 1
- Many

Note 0 and 1 are unusual (special) values. They might cause unanticipated behavior alone or in combination with particular values of other parameters.

Step 3: Introduce constraints

- A combination of values for each category corresponds to a test case specification
  - in the example we have 314,928 test cases
  - ***But*** most of which are impossible!
    - example
      zero slots and *at least one incompatible slot*
- Introduce constraints to
  - rule out impossible combinations
  - reduce the size of the test suite if too large

How is this number calculated?

Step 3: error constraint

[error] indicates a value class that

- corresponds to an erroneous values
- need be tried only once

Example

Model number: Malformed and Not in database

error value classes

- No need to test all possible combinations of errors
- One test is enough (we assume that handling an error case bypasses other program logic)

Example - Step 3: *error* constraint

Model number

Malformed	[error]
Not in database	[error]
Valid	

Correspondence of selection with model slots

Omitted slots	[error]
Extra slots	[error]
Mismatched slots	[error]
Complete correspondence	

Number of required comp. with non empty selection

0	[error]
< number of required slots	[error]

Required comp. selection

≥ 1 not in database	[error]

Number of models in database (#DBM)

0	[error]

Number of components in database (#DBC)

0	[error]

> Error constraints reduce test suite from 314,928 to 2,711 test cases

Step 3: *property* constraints

constraint [property] [if-property] rule out invalid combinations of values

[property] groups values of a single parameter to identify subsets of values with common properties

[if-property] bounds the choices of values for a category that can be combined with a particular value selected for a different category

Example

combine

Number of required comp. with non empty selection = number required slots [if RSMANY]

only with

Number of required slots for selected model (#SMRS) = Many [RSMANY]

Step 3: *property* constraints

constraint [property] [if-property] rule out invalid combinations of values

[property] groups values of a single parameter to identify subsets of values with common properties

[if-property] bounds the choices of values for a category that can be combined with a particular value selected for a different category

Example (see next page)

combine

Number of optional comp. with non empty selection

< number optional slots *[if OSNE]*

only with one of the following:

Number of optional slots for selected model (#SMOS)

1	[property OSNE]
Many	[property OSNE]

Example - Step 3: property constraints

Number of required slots for selected model (#SMRS)

1	[property RSNE]
Many	[property RSNE] [property RSMANY]

Number of optional slots for selected model (#SMOS)

1	[property OSNE]
Many	[property OSNE] [property OSMANY]

Number of required comp. with non empty selection

0	[if RSNE] [error]
< number required slots	[if RSNE] [error]
= number required slots	[if RSMANY]

Number of optional comp. with non empty selection

< number optional slots	[if OSNE]
= number optional slots	[if OSMANY]

from 2,711 to 908 test cases

Step 3 (cont): *single* constraints

[single] indicates a value class that test designers choose to test only once to reduce the number of test cases

Example

value some default for required component selection and optional component selection may be tested only once despite not being an erroneous condition

note -

single and error have the same effect but differ in rationale. Keeping them distinct is important for documentation and regression testing. *The former is usually solely for reducing the test case number but may not be based on "real constraint".*

Example - Step 3: *single* constraints

Number of required slots for selected model (#SMRS)

0	[single]
1	[property RSNE] [single]

Number of optional slots for selected model (#SMOS)

0	[single]
1	[single] [property OSNE]

Required component selection

Some default	[single]

Optional component selection

Some default	[single]

Number of models in database (#DBM)

1	[single]

Number of components in database (#DBC)

1	[single]

from 908 to 69 test cases

Check configuration – Summary

Parameter Model

- Model number
  - Malformed [error]
  - Not in database [error]
  - Valid
- Number of required slots for selected model (#SMRS)
  - 0 [single]
  - 1 [property RSNE] [single]
  - Many [property RSNE] [property RSMANY]
- Number of optional slots for selected model (#SMOS)
  - 0 [single]
  - 1 [property OSNE] [single]
  - Many [property OSNE] [property OSMANY]

Environment Product data base

- Number of models in database (#DBM)
  - 0 [error]
  - 1 [single]
  - Many
- Number of components in database (#DBC)
  - 0 [error]
  - 1 [single]
  - Many

Parameter Component

- Correspondence of selection with model slots
  - Omitted slots [error]
  - Extra slots [error]
  - Mismatched slots [error]
  - Complete correspondence
- # of required components (selection $\neq$ empty)
  - 0 [if RSNE] [error]
  - < number required slots [if RSNE] [error]
  - = number required slots [if RSMANY]
- Required component selection
  - Some defaults [single]
  - All valid
  - $\geq$ 1 incompatible with slots
  - $\geq$ 1 incompatible with another selection
  - $\geq$ 1 incompatible with model
  - $\geq$ 1 not in database [error]
- # of optional components (selection $\neq$ empty)
  - 0
  - < #SMOS [if OSNE]
  - = #SMOS [if OSMANY]
- Optional component selection
  - Some defaults [single]
  - All valid
  - $\geq$ 1 incompatible with slots
  - $\geq$ 1 incompatible with another selection
  - $\geq$ 1 incompatible with model
  - $\geq$ 1 not in database [error]

An excerpt of test case specifications derived from the value classes

Model#	# required slots	# optional slots	# Corr. w/ model slots	# required components	# optional components	Required components selection	Optional components selection	# Models in DB	# Components in DB	Exp result
malformed	many	many	same	EQR	0	all valid	all valid	many	many	Err
Not in DB	many	many	same	EQR	0	all valid	all valid	many	many	Err
valid	0	many	same	-	0	all valid	all valid	many	many	Accept
...					...					...
valid	many	many	same	EQR	EQO	in-other	in-mod	many	many	Reject
valid	many	many	same	EQR	EQO	in-mod	all valid	many	many	Reject
valid	many	many	same	EQR	EQO	in-mod	in-slot	many	many	Reject
valid	many	many	same	EQR	EQO	in-mod	in-other	many	many	Reject
valid	many	many	same	EQR	EQO	in-mod	in-mod	many	many	Reject

Legend

EQR	= # req slot
EQO	= # opt slot
in-mod	$\geq$1 incompat w/ model
in-other	$\geq$1 incompat w/ another slot
in-slot	$\geq$1 incompat w/ slot

Next ...

- Category partition testing gave us
  - Systematic approach: Identify characteristics and values (the creative step), generate combinations (the mechanical step)
- But …
  - Test suite size grows very rapidly with number of categories. Can we use a non-exhaustive approach?
- Pairwise (and n-way) combinatorial testing do
  - Combine values systematically but not exhaustively
  - Rationale: Most unplanned interactions are among just two or a few parameters or parameter characteristics

Pairwise combinatorial testing

- Category partition works well when intuitive constraints reduce the number of combinations to a small amount of test cases
  - Without many constraints, the number of combinations may be unmanageable
- Pairwise combination (instead of exhaustive)
  - Generate combinations that efficiently cover all pairs (triples,…) of classes
  - Rationale: most failures are triggered by single values or combinations of a few values. Covering pairs (triples,…) reduces the number of test cases, but reveals most faults

Example: Display Control

No constraints reduce the total number of combinations
432 (3x4x3x4x3) test cases
if we consider all combinations

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	

Pairwise combinations: 17 test cases

Language	Color	Display Mode	Fonts	Screen Size
English	Monochrome	Full-graphics	Minimal	Hand-held
English	Color-map	Text-only	Standard	Full-size
English	16-bit	Limited-bandwidth	-	Full-size
English	True-color	Text-only	Document-loaded	Laptop
French	Monochrome	Limited-bandwidth	Standard	Laptop
French	Color-map	Full-graphics	Document-loaded	Full-size
French	16-bit	Text-only	Minimal	-
French	True-color	-	-	Hand-held
Spanish	Monochrome	-	Document-loaded	Full-size
Spanish	Color-map	Limited-bandwidth	Minimal	Hand-held
Spanish	16-bit	Full-graphics	Standard	Laptop
Spanish	True-color	Text-only	-	Hand-held
Portuguese	-	-	Monochrome	Text-only
Portuguese	Color-map	-	Minimal	Laptop
Portuguese	16-bit	Limited-bandwidth	Document-loaded	Hand-held
Portuguese	True-color	Full-graphics	Minimal	Full-size
Portuguese	True-color	Limited-bandwidth	Standard	Hand-held

Adding constraints

- Simple constraints

  example: color monochrome not compatible with screen laptop and full size

  can be handled by considering the case in separate tables

Example: Monochrome only with hand-held

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	
limited-bandwidth	Spanish	Document-loaded	16-bit	
	Portuguese		True-color	

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal		
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	

Next ...

- Category-partition approach gives us …
  - Separation between (manual) identification of parameter characteristics and values and (automatic) generation of test cases that combine them
  - Constraints to reduce the number of combinations
- Pairwise (or n-way) testing gives us …
  - Much smaller test suites, even without constraints
    - (but we can still use constraints)
- We still need …
  - Help to make the manual step more systematic

Catalog based testing

- Deriving value classes requires human judgment
- Gathering experience in a systematic collection can:
  - speed up the test design process
  - routinize many decisions, better focusing human effort
  - accelerate training and reduce human error
- Catalogs capture the experience of test designers by listing important cases for each possible type of variable
  - *Example: if the computation uses an integer variable, a catalog might indicate the following relevant cases*
    - *The element immediately preceding the lower bound*
    - *The lower bound of the interval*
    - *A non-boundary element within the interval*
    - *The upper bound of the interval*
    - *The element immediately following the upper bound*

Catalog based testing process

Step1:

Analyze the initial specification to identify simple elements:

- Pre-conditions
- Post-conditions
- Definitions
- Variables
- Operations

Step 2:

Derive a first set of test case specifications from pre-conditions, post-conditions and definitions

Step 3:

Complete the set of test case specifications using test catalogs

An informal specification: cgi_decode

Function *cgi_decode* translates a cgi-encoded string to a plain ASCII string, reversing the encoding applied by the common gateway interface (CGI) of most web servers

CGI translates spaces to +, and translates most other non-alphanumeric characters to hexadecimal escape sequences

cgi_decode maps + to spaces, %xy (where x and y are hexadecimal digits) to the corresponding ASCII character, and other alphanumeric characters to themselves

An informal specification: input/output

[INPUT] encoded: string of characters (the input CGI sequence)

can contain:

- alphanumeric characters
- the character +
- the substring %xy, where x and y are hexadecimal digits

is terminated by a null character

[OUTPUT] decoded: string of characters (the plain ASCII characters corresponding to the input CGI sequence)

- alphanumeric characters copied into output (in corresponding positions)
- blank for each + character in the input
- single ASCII character with value xy for each substring %xy

[OUTPUT] return value cgi_decode returns

- 0 for success
- 1 if the input is malformed

Step 1: Identify simple elements

Pre-conditions: conditions on inputs that must be true before the execution

- validated preconditions: checked by the system
- assumed preconditions: assumed by the system

Post-conditions: results of the execution

Variables: elements used for the computation

Operations: main operations on variables and inputs

Definitions: abbreviations

Step 1: cgi_decode (var, def, op.)

VAR 1 encoded: a string of ASCII characters

VAR 2 decoded: a string of ASCII characters

VAR 3 return value: a boolean

DEF 1 hexadecimal characters, in range ['0' .. '9', 'A' .. 'F', 'a' .. 'f']

DEF 2 sequences %xy, where x and y are hexadecimal characters

DEF 3 CGI items as alphanumeric character, or '+', or CGI hexadecimal

OP 1 Scan encoded

Step 1: cgi_decode (pre and post)

PRE 1 (Assumed) input string encoded null-terminated string of chars

PRE 2 (Validated) input string encoded sequence of CGI items

POST 1 if encoded contains alphanumeric characters, they are copied to the output string

POST 2 if encoded contains characters +, they are replaced in the output string by ASCII SPACE characters

POST 3 if encoded contains CGI hexadecimals, they are replaced by the corresponding ASCII characters

POST 4 if encoded is processed correctly, it returns 0

POST 5 if encoded contains a wrong CGI hexadecimal (a substring xy, where either x or y are absent or are not hexadecimal digits, cgi_decode returns 1

POST 6 if encoded contains any illegal character, it returns 1

Step 2: Derive initial set of test case specs

- Validated preconditions:
  - simple precondition (expression without operators)
    - 2 classes of inputs:
      - inputs that satisfy the precondition
      - inputs that do not satisfy the precondition
  - compound precondition (with AND or OR):
    - apply modified condition/decision (MC/DC) criterion
- Assumed precondition:
  - apply MC/DC only to “OR preconditions”
- Postconditions and Definitions :
  - if given as conditional expressions, consider conditions as if they were validated preconditions

Step 2: cgi_decode (tests from Pre)

PRE 2 (Validated) the input string encoded is a sequence of CGI items

- *TC-PRE2-1: encoded is a sequence of CGI items*
- *TC-PRE2-2: encoded is not a sequence of CGI items*

POST 1 if encoded contains alphanumeric characters, they are copied in the output string in the corresponding position

- *TC-POST1-1: encoded contains alphanumeric characters*
- *TC-POST1-2: encoded does not contain alphanumeric characters*

- POST 2 if encoded contains characters +, they are replaced in the output string by ASCII SPACE characters
  - *TC-POST2-1: encoded contains character +*
  - *TC-POST2-2: encoded does not contain character +*

Step 2: cgi_decode (tests from Post)

POST 3 if encoded contains CGI hexadecimals, they are replaced by the corresponding ASCII characters

- *TC-POST3-1 Encoded: contains CGI hexadecimals*
- *TC-POST3-2 Encoded: does not contain a CGI hexadecimal*

POST 4 if encoded is processed correctly, it returns 0

POST 5 if encoded contains a wrong CGI hexadecimal (a substring xy, where either x or y are absent or are not hexadecimal digits, cgi_decode returns 1

- *TC-POST5-1 Encoded: contains erroneous CGI hexadecimals*

POST 6 if encoded contains any illegal character, it returns 1

- *TC-POST6-1 Encoded: contains illegal characters*

Step 2: cgi_decode (tests from Var)

VAR 1 encoded: a string of ASCII characters

VAR 2 decoded: a string of ASCII characters

VAR 3 return value: a boolean

DEF 1 hexadecimal characters, in range ['0' .. '9', 'A' .. 'F', 'a' .. 'f']

DEF 2 sequences %xy, where x and y are hexadecimal characters

DEF 3 CGI items as alphanumeric character, or '+', or CGI hexadecimal

OP 1 Scan encoded

Step 3: Apply the catalog

- Scan the catalog sequentially
- For each element of the catalog
  - scan the specifications
  - apply the catalog entry
- Delete redundant test cases
- Catalog:
  - List of kinds of elements that can occur in a specification
  - Each catalog entry is associated with a list of generic test case specifications

  Example:

  catalog entry Boolean

  two test case specifications: true, false

  Label in/out indicate if applicable only to input, output, both

A simple catalog (part I)

• Boolean	
– True	in/out
– False	in/out
• Enumeration	
– Each enumerated value	in/out
– Some value outside the enumerated set	in
• Range L … U	
– L-1	in
– L	in/out
– A value between L and U	in/out
– U	in/out
– U+1	in
• Numeric Constant C	
– C	in/out
– C –1	in
– C+1	in
– Any other constant compatible with C	in

A simple catalog (part II)

- Non-Numeric Constant C
  - C — in/out
  - Any other constant compatible with C — in
  - Some other compatible value — in
- Sequence
  - Empty — in/out
  - A single element — in/out
  - More than one element — in/out
  - Maximum length (if bounded) or very long — in/out
  - Longer than maximum length (if bounded) — in
  - Incorrectly terminated — in
- Scan with action on elements P
  - P occurs at beginning of sequence — in
  - P occurs in interior of sequence — in
  - P occurs at end of sequence — in
  - PP occurs contiguously — in
  - P does not occur in sequence — in
  - pP where p is a proper prefix of P — in
  - Proper prefix p occurs at end of sequence — in

Example - Step 3: Catalog entry *boolean*

- Boolean
  - True in/out
  - False in/out

applies to *return value*
generates 2 test cases already covered by
TC-PRE2-1 and TC-PRE2-2

Example - Step 3: entry *enumeration*

- Enumeration
  - Each enumerated value in/out
  - Some value outside the enumerated set in

applies to

- *CGI item (DEF 3)*

  included in TC-POST1-1, TC-POST1-2, TC-POST2-1, TC-POST2-2, TC-POST3-1, TC-POST3-2

Example - Step 3: entry *enumeration*

applies also to improper CGI hexadecimals

- New test case specifications
  - TC-POST5-2 *encoded terminated with %x, where x is a hexadecimal digit*
  - TC-POST5-3 *encoded contains %ky, where k is not a hexadecimal digit and y is a hexadecimal digit*
  - TC-POST5-4 *encoded contains %xk, where x is a hexadecimal digit and k is not*
- Old test case specifications can be eliminated if they are less specific than the newly generated cases
  - TC-POST3-1 *encoded contains CGI hexadecimals*
  - TC-POST5-1 *encoded contains erroneous CGI hexadecimals*

Example - Step 3: entry *range*

Applies to variables defined on a finite range

- hexadecimal digit
  - characters / and :
    (before 0 and after 9 in the ASCII table)
  - values 0 and 9 (bounds),
  - one value between 0 and 9
  - @, G, A, F, one value between A and F
  - }, g, a, f, one value between a and f
  - *30 new test cases* (15 for each character)
- Alphanumeric char (DEF 3):
  - *5 new test cases*

Example - Step 3: entries *numeric* and *non-numeric constant*

Numeric Constant does not apply

Non-Numeric Constant applies to
+ and %, in DEF 3 and DEF 2:
- *6 new Test Cases* (all redundant)

Step 3: entry *sequence*

apply to

encoded (VAR 1), *decoded* (VAR 2), and *cgi-item* (DEF 2)

- 6 new Test Cases for each variable
- Only 6 are non-redundant:
  - *encoded*
    - empty sequence
    - sequence of length one
    - long sequence
  - *cgi-item*
    - % terminated sequence (subsequence with one char)
    - % initiated sequence
    - sequence including %xyz, with x, y, and z hexadecimals

Step 3: entry *scan*

applies to *Scan encoded* (OP 1) and generates 17 test cases:

- only 10 are non-redundant

summary of generated test cases (i/ii)

TC-POST2-1: *encoded* contains *+*
TC-POST2-2: *encoded* does not contain *+*
TC-POST3-2: *encoded* does not contain a CGI-hexadecimal
TC-POST5-2: *encoded* terminated with *%x*
TC-VAR1-1: *encoded* is the empty sequence
TC-VAR1-2: *encoded* a sequence containing a single character
TC-VAR1-3: *encoded* is a very long sequence
TC-DEF2-1: *encoded* contains *%/y*
TC-DEF2-2: *encoded* contains *%0y*
TC-DEF2-3: *encoded* contains '*%xy'* (x in [1..8])
TC-DEF2-4: *encoded* contains '*%9y'*
TC-DEF2-5: *encoded* contains '*%:y'*
TC-DEF2-6: *encoded* contains '*%@y'*
TC-DEF2-7: *encoded* contains '*%Ay'*
TC-DEF2-8: *encoded* contains *'%xy'* (x in [B..E])
TC-DEF2-9: *encoded* contains '*%Fy'*
TC-DEF2-10: *encoded* contains '*%Gy'*
TC-DEF2-11: *encoded* contains *%`y'*
TC-DEF2-12: *encoded* contains *%ay*
TC-DEF2-13: *encoded* contains *%xy* (*x* in [*b..e*])
TC-DEF2-14: *encoded* contains *%fy'*
TC-DEF2-15: *encoded* contains *%gy*
TC-DEF2-16: *encoded* contains *%x/*
TC-DEF2-17: *encoded* contains *%x0*
TC-DEF2-18: *encoded* contains *%xy* (*y* in [*1..8*])
TC-DEF2-19: *encoded* contains *%x9*
TC-DEF2-20: *encoded* contains *%x:*
TC-DEF2-21: *encoded* contains *%x@*
TC-DEF2-22: *encoded* contains *%xA*
TC-DEF2-23: *encoded* contains *%xy* (*y* in [*B..E*])
TC-DEF2-24: *encoded* contains *%xF*
TC-DEF2-25: *encoded* contains *%xG*
TC-DEF2-26: *encoded* contains *%x`*
TC-DEF2-27: *encoded* contains *%xa*
TC-DEF2-28: *encoded* contains *%xy* (*y* in [*b..e*])
TC-DEF2-29: *encoded* contains *%xf*

Summary of generated test cases (ii/ii)

TC-DEF2-30: *encoded* contains *%xg*
TC-DEF2-31: *encoded* terminates with *%*
TC-DEF2-32: *encoded* contains *%xyz*
TC-DEF3-1: *encoded* contains */*
TC-DEF3-2: *encoded* contains *0*
TC-DEF3-3: *encoded* contains *c* in [*1..8*]
TC-DEF3-4: *encoded* contains *9*
TC-DEF3-5: *encoded* contains *:*
TC-DEF3-6: *encoded* contains *@*
TC-DEF3-7: *encoded* contains *A*
TC-DEF3-8: *encoded* contains *c* in *[B..Y]*
TC-DEF3-9: *encoded* contains *Z*
TC-DEF3-10: *encoded* contains *[*
TC-DEF3-11: *encoded* contains `
TC-DEF3-12: *encoded* contains *a*
TC-DEF3-13: *encoded* contains *c* in [*b..y*]
TC-DEF3-14: *encoded* contains *z*
TC-DEF3-15: *encoded* contains {

TC-OP1-1: *encoded* starts with an alphanumeric character
TC-OP1-2: *encoded* starts with *+*
TC-OP1-3: *encoded* starts with *%xy*
TC-OP1-4: *encoded* terminates with an alphanumeric character
TC-OP1-5: *encoded* terminates with *+*
TC-OP1-6: *encoded* terminated with *%xy*
TC-OP1-7: *encoded* contains two consecutive alphanumeric characters
TC-OP1-8: *encoded* contains *++*
TC-OP1-9: *encoded* contains *%xy%zw*
TC-OP1-10: *encoded* contains *%x%yz*

Summary

- From category partition testing:
  - Division into a (manual) step of identifying categories and values, with constraints, and an (automated) step of generating combinations
- From catalog based testing:
  - Improving the manual step by recording and using standard patterns for identifying significant values
- From pairwise testing:
  - Systematic generation of smaller test suites
- These ideas can be combined